View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarlyCommons@Penn

University of Pennsylvania
ScholarlyCommons

Departmental Papers (CIS) | Department of Computer & Information Science

4-27-2015

Verified ROS-Based Deployment of Platform-Independent Control Systems

Wenrui Meng
University of Pennsylvania

Junkil Park
University of Pennsylvania, park11@seas.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Stephanie Weirich
University of Pennsylvania, sweirich@cis.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_papers

 Part of the Computer Sciences Commons, and the Hardware Systems Commons

Recommended Citation

Wenrui Meng, Junkil Park, Oleg Sokolsky, Stephanie Weirich, and Insup Lee, "Verified ROS-Based Deployment of Platform-Independent Control Systems", *NASA Formal Methods* , 248-262. April 2015. http://dx.doi.org/10.1007/978-3-319-17524-9_18

7th NASA Formal Methods Symposium (NFM 2015), Pasadena, CA, April 27-29, 2015.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_papers/794
For more information, please contact libraryrepository@pobox.upenn.edu.

Verified ROS-Based Deployment of Platform-Independent Control Systems

Abstract
The paper considers the problem of model-based deployment of platform-independent control code on a specific platform. The approach is based on automatic generation of platform-specific glue code from an architectural model of the system. We present a tool, ROSGen, that generates the glue code based on a declarative specification of platform interfaces. Our implementation targets the popular Robot Operating System (ROS) platform. We demonstrate that the code generation process is amenable to formal verification. The code generator is implemented in Coq and relies on the infrastructure provided by the CompCert and VST tool. We prove that the generated code always correctly connects the controller function to sensors and actuators in the robot. We use ROSGen to implement a cruise control system on the LandShark robot.

Disciplines
Computer Engineering | Computer Sciences | Hardware Systems

Comments
7th NASA Formal Methods Symposium (NFM 2015), Pasadena, CA, April 27-29, 2015.

This conference paper is available at ScholarlyCommons: http://repository.upenn.edu/cis_papers/794

Verified ROS-Based Deployment of Platform-Independent Control Systems*

Wenrui Meng, Junkil Park, Oleg Sokolsky, Stephanie Weirich, and Insup Lee

University of Pennsylvania

Abstract. The paper considers the problem of model-based deployment of platform-independent control code on a specific platform. The approach is based on automatic generation of platform-specific glue code from an architectural model of the system. We present a tool, ROSGen, that generates the glue code based on a declarative specification of platform interfaces. Our implementation targets the popular Robot Operating System (ROS) platform. We demonstrate that the code generation process is amenable to formal verification. The code generator is implemented in Coq and relies on the infrastructure provided by the CompCert and VST tool. We prove that the generated code always correctly connects the controller function to sensors and actuators in the robot. We use ROSGen to implement a cruise control system on the LandShark robot.

1 Introduction

Modern cyber-physical systems are typically constructed from individually developed components. This process involves two steps: first, developing the components in a platform independent way, and second, deploying these components on a specific architecture, using a middleware platform to implement the connections between the components.

Model-based development aids in both parts of this development process. First, in developing individual components, component behaviors are abstractly specified by data models, state charts, or diagrams. These diagrams can be expressed using design tools such as Simulink/Stateflow , UPPAAL [1], or SCADE/Lustre [2]. Code generation tools then convert these diagrams into code, typically platform-independent C source code. This generative approach helps us to preserve properties verified at the modeling level, making sure that component implementations also satisfy these properties.

Second, system architectural models describe the relationships between the components of the system. For example, in an autonomous robotic system the architectural model specifies (1) how each component should be executed (such as how the periodic execution within a given period may be specified), (2) how

*This research is supported in part by DARPA HACMS program under agreement FA8750-12-2-0247. The views expressed are those of the authors and may not reflect the official policy or position of the Department of Defense or the U.S. Government.

system inputs, such as sensor streams, should be routed to inputs of components processing the streams, and (3) how outputs of each component should be routed to inputs of other components or to system outputs (such as actuators). A significant part of platform configuration is providing a platform-specific wrapper for the platform-independent component implementation. The wrapper (also known as the glue code) uses platform APIs to schedule component execution, to obtain inputs for the component, and to forward its outputs. A faulty deployment undermines the benefits of provably correct implementation of individual components. Platform configurations, therefore, should be automatically generated from the architectural model to ensure correct integration of individual components.

In this paper, we address the problem of automatically generating provably correct glue code for a particular deployment platform from a given architectural model. We use the Robot Operating System (ROS)[1] as our target platform, a "thin, message-based, peer-to-peer" [3] robotics middleware designed for mobile manipulators. The ROS platform has recently gained popularity in the robotics community because it raises the level of abstraction in embedded control system development. ROS-based applications are assembled from multiple ROS nodes that run concurrently. ROS supports communication between these nodes using a publish/subscribe-based message system.

To that end, we develop a ROS glue code generator, called ROSGen, that automatically generates such glue code from system architecture specifications. The input language for our code generator is a domain-specific language, called a `ROS node model`, that specifies the ROS nodes that comprise the system and ROS topics that the nodes subscribe to and publish on.

Of course, by generating code we eliminate some sources of programmer error in system development. However, for safety critical systems, we want the highest level of assurance. We would like to prove that the output of our code generator satisfies strong correctness and safety requirements. One can take two approaches for the verification of generated code; first, one may verify every output individually. Alternatively, which is generally much harder, one may verify the code generator itself.

Our code generator is designed to support (both forms of) formal verification. ROSGen is implemented using the Coq proof assistant [4], making the full higher-order logic of Coq available for reasoning about both the output of the generator (represented as a Coq data structure) and the code generator itself (represented as a Coq function). In this context, we have used both approaches for verification.

We have applied ROSGen as part of a case study of glue code generation for the Black-i Robotics LandShark platform. The LandShark is an unmanned ground vehicle typically used to extend human capabilities, often in dangerous environments such as at a chemical spill or for sentry duty. ROSGen can generate glue code for this platform, and we have proven that the generated code satisfies a crucial `Data Delivery Correctness (DDC)` property: that the arriving sensor message will be correctly delivered to the control function and that the output of

[1] www.ros.org

the control function will be correctly delivered to the actuators. We express and prove this property using the Verified Software Toolchain (VST) tool [5], which provides a higher-order separation logic for reasoning about memory usage in C programs. Our proof has been mechanically checked by Coq.

Moreover, we prove the generalized `DDC` property of the code generator itself. That is, we can show that `every` output of ROSGen satisfies the same `DDC` property that we have shown for the LandShark instance. In general, this is a hard problem. However, in our case, because of the relatively simple code structure and because the property of interest is concerned with data transfer, we can generalize the proof of instances of the generated code to the proof of the generator itself.

In summary, this paper makes the following contributions:

- We introduce a domain specific language for describing the ROS nodes. We develop a code generator ROSGen to generate the robotics glue code according to a given ROS node model (Section 4).
- We demonstrate an application of ROSGen to a case study of a robotic control system and prove, using a suite of Coq-based tools, that the glue code correctly delivers data according to the ROS node model of the controller (Section 5).
- Finally, we verify that, given a well-formed ROS node model, ROSGen always generates code that satisfies the data delivery correctness property (Section 6).

The rest of the paper is organized as follows: we introduce the relevant work that our code generation is dependent on in Section 2. Section 3 explains the architecture of the ROS based control system and introduces the LandShark case study. In Sections 4, 5 and 6, we explain our code generation approach for ROS based control system and the verification for the generated code and code generator itself. We discuss related work in Section 7 and conclude in Section 8.

The Coq implementation of the code generator and relevant parts of case study can be downloaded from `http://rtg.cis.upenn.edu/HACMS/codegen.html`. The technical report [6] full version of this paper, including the formal VST specification and proof of DDC property, can also be found on the same webpage.

2 Proof Environment

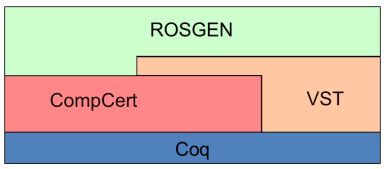

Fig. 1. ROSGen dependency structure

Figure 1 shows the tools underlying ROSGen, which are briefly described below.

Coq. The Coq Proof Assistant[2] is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs.
CompCert. CompCert [7] is a formally verified optimizing compiler for the C programming language that currently targets PowerPC, ARM and 32-bit x86 architectures. The compiler is specified, implemented and proved correct using the Coq proof assistant. It targets embedded systems programming, with stringent reliability requirements. CompCert's source language, a large subset of C called Clight, is the target language of our code generator; our generator produces abstract syntax values for Clight.

The formal semantics of Clight is mechanized using Coq. It supports many types including integral types (integers and floats in various sizes and signedness), array types, pointer types (including pointers to functions), function types, as well as struct and union types. A Clight program is composed of a list of declarations for global variables (name and type), a list of functions and an identifier naming the entry point of the program (the main function in C). *Verified software toolchain.* The goal of the Verified Software Toolchain (VST)[3] project is to verify that the assertions claimed at the top of a software toolchain really hold in the machine language program, running in the operating system context, on a weakly-consistent-shared-memory machine. It defines `Verifiable C`, a higher-order concurrent separation logic for Clight. `Verifiable C` has been proven sound with respect to the operational semantics of CompCert C [5].

The `Verifiable C` program logic extends Hoare logic by including separation logic constructs to support reasoning about mutable data structures such as arrays and pointers. In separation logic, an assertion holds on a particular subheap and assertions on different subheap are independent. As a result logical reasoning is modular. VST provides a tactic system for proving correctness properties, specified by the VST assertions, of C light programs. The most significant of these are the `forward` tactic, which symbolically executes the code, and the `entailer` tactic, which simplifies and often solves VST assertions [8].

3 ROS-based control system

3.1 Robot operating system

ROS is a widely used component-based middleware for robotic system applications. A software component in ROS is called a ROS node. A ROS application usually consists of multiple ROS nodes running concurrently. The ROS nodes asynchronously communicate with each other. Communication in ROS is based on the Publish/Subscribe paradigm and uses structured message types. ROS Services are the mechanism to implement remote procedure calls in ROS, which are synchronous and blocking.

[2] http://coq.inria.fr/
[3] http://vst.cs.princeton.edu/

```
void callback(MessageType msg) { ... };
main(){
  Subscribe(..., callback);
  Advertise(...);
  while( ros_ok() ){
    SpinOnce();
      /* Process the input to the controller */
    Controller_step();
      /* Process the output of the controller */
    Publish(...); }}
```

Fig. 2. ROS-based controller system skeleton

Figure 2 shows the skeleton of a ROS-based control system. In order to subscribe to a topic in ROS, users need to define a callback function. A callback function for a topic is a message handler that is invoked to process the new messages when they arrive. `Subscribe` is a function from the ROS API that registers subscription information: a topic name, the message type, the internal buffer size and the callback function for those messages. If a new message is received, it is stored in an internal buffer. It replaces the oldest message in the buffer if the buffer is already full. When the ROS API function `SpinOnce` is invoked, all registered callback functions are invoked for every message in the internal buffers. In order to publish a topic in ROS, users should use the ROS API function `Advertise` to first create a publisher with a topic name, message type, and internal buffer size. The ROS API `Publish` function is then used to publish a message.

3.2 Case study of LandShark control system

Fig. 3. LandShark robot

In this section we illustrate a typical ROS-based control system using the LandShark robot. The LandShark is an electric unmanned ground vehicle, shown in Figure 3, manufactured by Black-I Robotics.[4] Our case study develops a

[4] http://www.blackirobotics.com/

constant-speed cruise control algorithm that is resilient to attacks on vehicle sensors. The LandShark uses three sensors, GPS, a left wheel encoder and a right wheel encoder, to estimate its current velocity. These sensors can be compromised by attacks, such as GPS spoofing, that cause confusion in estimating the current velocity of the vehicle. The attack-resilient cruise controller of LandShark uses multiple independent sensors and the knowledge of the system model in order to correctly estimate the current velocity of the vehicle and drive the vehicle with a given constant velocity [9].

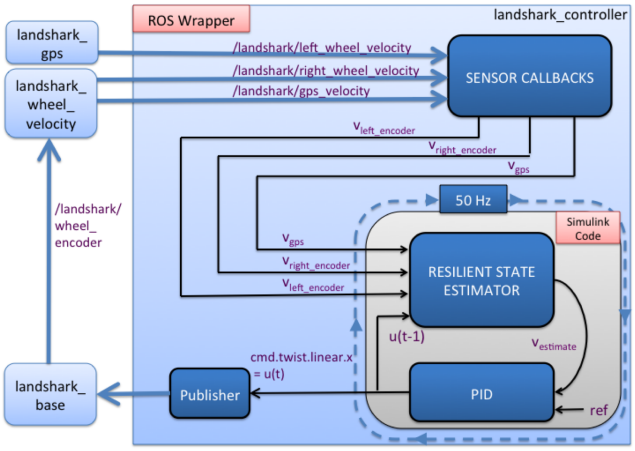

Fig. 4. LandShark control system architecture

Figure 4 shows the architecture of the LandShark control system, which consists of sensor/actuation/controller nodes and the connections between them through topic-based pub/sub communication. The ROS nodes landshark_gps and landshark_base are associated with sensors that read GPS and wheel encoder values respectively and publish them. The ROS node landshark_wheel_velocity subscribes to the series of wheel encoder values and publishes the velocity of the vehicle calculated from them. The ROS node landshark_base also plays a role as an actuation node in that it subscribes to the actuation commands and actuates the vehicle according to them. The ROS node landshark_controller is the controller node that subscribes to sensor value messages and publishes actuation commands. The landshark_controller node is periodically invoked at the rate of 50 Hz to execute the Simulink-generated step function. In each invocation, the callback functions are invoked by `SpinOnce` to process the messages received. The callback functions store the sensor messages in global variables. The sensor values in the global variables are transferred to the input data structure of the control algorithm function that is generated by Simulink. The step function is executed to calculate the actuation command, which is encapsulated in a ROS message variable and published by the publisher.

4 Code Generation

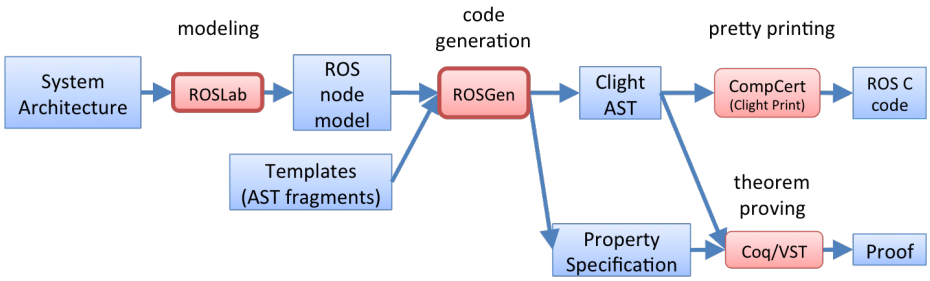

Fig. 5. Verified code generation toolchain

Our toolchain for verified code generation appears in Figure 5. The ROSLab tool supports the design of system architectures, allowing the creation of a diagram block using a graphical user interface. The diagram block in ROSLab can then be exported in our architectural description language as a ROS node model. With the ROS node model, ROSGen produces an abstract syntax tree for a subset of C called Clight, by instantiating a Clight AST template. In addition, ROSGen also generates a VST specification for each function, describing its Data Delivery Correctness `DDC` properties. We can prove that the generated code satisfies these specifications, as we demonstrate in Section 6. The final C code, which is run on the LandShark, is produced by the CompCert compiler using its pretty printer.

ROSLab tool. ROSLab is a modular programming environment for robotic applications based on ROS. ROSLab enables users to model an architecture of a ROS application that consists of a set of ROS nodes and the connections between them. The interfaces of some commonly used ROS nodes such as sensor and actuator nodes are pre-defined in ROSLab. Users can define a new ROS node and its interface by selecting the pub/sub channels to add to the interface of the node.

4.1 ROS node model

A diagram block in ROSLab can be exported as a ROS node model. A ROS node model includes the period at which the node is to be invoked; the list of topics that the node publishes or subscribes to; the name and the I/O interface of the controller function that the node will run; and finally, a mapping from subscribed and published topics to inputs and outputs of the controller function.

The ROS node model for the `landshark_controller` ROS node in Figure 4 is shown in Table 1. The name of the node, the period of the controller, and the name of the controller function that the node will execute are shown at the top of the table. Published topics are indicated by the letter P and subscribed topics are indicated by the letter S. For each topic, the unique topic name and the

Node Information				
period	node name		controller name	
20	landshark_controller		Controller	
ROS Topics				
type	topic name	message package	message type	buffer size
S	/landshark/left_wheel_velocity	geometry_msgs	TwistStamped	1
S	/landshark/right_wheel_velocity	geometry_msgs	TwistStamped	1
S	/landshark/gps_velocity	geometry_msgs	TwistStamped	1
P	/landshark_control/base_velocity	geometry_msgs	TwistStamped	1
Controller Interface				
I/O	name	record type		
I	Controller_U	(In1, double), (In2, double), (In3, double)		
O	Controller_Y	(Out1, double)		
Interface Relation				
type	topic		controller	
SI	/landshark/left_wheel_velocity, twist, linear, x		Controller_U, In1	
SI	/landshark/right_wheel_velocity, twist, linear, x		Controller_U, In2	
SI	/landshark/gps_velocity, twist, linear, x		Controller_U, In3	
PO	/landshark_control/base_velocity, twist, linear, x		Controller_Y, Out1	

Table 1. ROS node model for LandShark

type of messages are given. Next, the ROS node model specifies the controller function interface. In our case study, the controller function is generated from a Simulink model of the controller, and the names and types of input and output variables are following the Simulink code generator conventions. Finally, the interface relation represents the mapping from relevant fields of subscribed sensor messages to the fields in the input data structures of the controller function, and similarly for outputs of the controller function to published actuator messages.

4.2 ROSGen

Symbol table. As the first step in code generation, ROSGen constructs a Coq data structure representing symbols to be used in the generated code. The names are obtained by parsing the ROS node model. Types for the controller function interface are given in the node model. Types for ROS messages referenced in the node model are obtained by parsing the corresponding C header files.
Code templates. Code generation proceeds by instantiating templates that are Clight AST fragments. We use a top-level template, representing the whole program, and a set of local templates. The top-level template is shown in Figure 6. The program contains a list of global definitions and the name for the main function. A global definition can be either a variable definition or a function definition. One of the global definitions is the definition of the main function, which is partially constructed in the top-level template. Light-colored triangles in the top-level template represent holes that are filled with instantiations of local templates. Local templates are used to capture global definitions, such as callback function definitions, global variables used to transfer data from callback

functions to the main function, and also glue code functions explained in more detail below. Holes in local templates can represent statements, as well as variable ids and types that are filled with references to the symbol table. Once all the templates are instantiated, the final C code is produced by CompCert pretty printing.

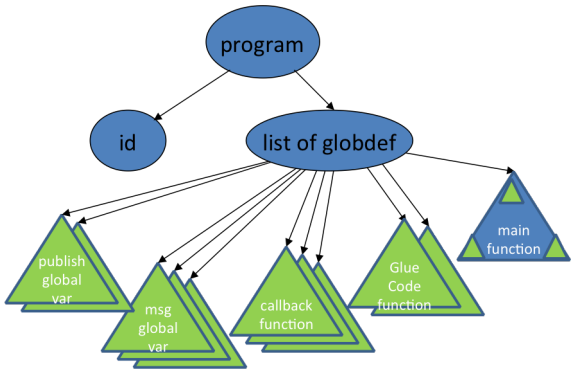

Fig. 6. Top-level template

To make proofs more efficient, we modularize the body of the main function from Figure 2 into several functions. The while loop is encapsulated as a loop function. Within the loop function, we wrap the code for transferring data from global variable to controller input and controller output to publish input as input_glue and output_glue function, respectively. Figure 7 shows the generated code for the glue functions.

```
void input_glue(){
  double temp;
  temp = landshark_left_encoder_velocity_msg.twist.linear.x;
  Controller_U.In1 = temp;
  temp = landshark_right_encoder_velocity_msg.twist.linear.x;
  Controller_U.In2 = temp;
  temp = landshark_gps_velocity_msg.twist.linear.x;
  Controller_U.In3 = temp;
  return;}
```

Fig. 7. Input glue function

5 Code Proof

We use VST to prove a DDC property for the generated Clight AST. Because VST is based on axiomatic semantics, we specify the DDC properties with pre-

and post-conditions that capture the relation between the origin variables and destination variables.

5.1 Data delivery correctness property of glue code

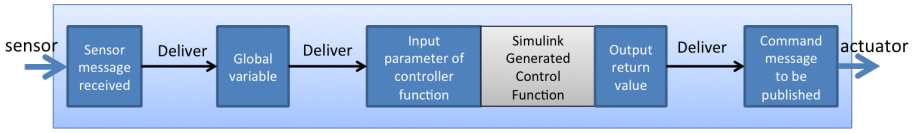

Fig. 8. Data delivery correctness property for ROS-based control system

The main purpose of the ROS glue code is linking the sensor input, controller function and actuator, so the critical property of glue code should capture the correctness of the linking. In ROS glue code, the linking correctness means that the sensor message is delivered into controller function input correctly. In addition, the output of the controller function correctly is delivered into the actuator input. We specify the linking correctness property of the ROS glue code as a `DDC Property`. This property indicates that the information from the origin should be consistent with the system specification when it arrives at the destination. For example, we design the system in the way that the sensor message is directly stored into global variables. So the `DDC` property of this operation is that the original value of the sensor message is equal to the value of the updated global variable. If we need to transform the original value, then the `DDC` property should specify the relation between the original value and destination value according to the transformation.

5.2 Generating function specifications

ROSGen automatically generates VST function specifications according to the ROS node model for both generated functions and ROS API functions. In VST, users specify properties through function specifications, so we wrap our glue code as functions. These functions include callback, and input and output glue functions for the controller step function.

As shown in Figure 8, the specifications of the functions capture the `DDC` property of the generated AST instance. The callback functions are responsible for transferring sensor messages to global message variables; the input glue function is responsible for transferring global message variable to the input parameter of controller function; and the output glue function is responsible for transferring output of controller function to the parameters of publish function. For each part, the `DDC` property specification defines the precondition that the original value is stored in memory and the postcondition that the destination contains the desired value according to the original value.

As shown in Figure 9, the `input_glue` function has the precondition that there are three global message variables with values and controller input `Controller_U`

with an unknown value. The postcondition indicates that `Controller_U` contains the right value from corresponding fields defined in the ROS node model and that the values of those three global variables are unchanged. By satisfying this postcondition, we can guarantee that the input to the controller function is consistent to the architecture ROS node model.

```
Precondition:
   {landshark_left_encoder_velocity_msg ← data1,
   landshark_right_encoder_velocity_msg ← data2,
   landshark_gps_velocity_msg ← data3,
   Controller_U ← _}
Postcondition:
   {landshark_left_encoder_velocity_msg ← data1,
   landshark_right_encoder_velocity_msg ← data2,
   landshark_gps_velocity_msg ← data3,
   Controller_U ← {data1.twist.linear.x, data2.twist.linear.x, data3.twist.linear.x}}
```

Fig. 9. DDC specification of input glue function

Specification of ROS API functions. For the code proof, we have to supply specifications of ROS API functions called by the code. These specifications are treated as assumptions in the proof. Here, specification of the ROS API function `SpinOnce` presents a challenge. The function implicitly invokes the registered callback functions to update global variables with new sensor values. The straightforward way to specify `SpinOnce` is to refer to the specifications of the callbacks. However, currently, VST does not support using other function specifications to construct a specification. Therefore, we specify the `SpinOnce` function using the global variables update essentially incorporating callback specifications directly into the `SpinOnce` specification. This specification has the precondition that the global variables are stored somewhere of memory and the postcondition that the global variables are updated to the provided data.

5.3 Code proof strategy

We use the tactics from VST proof automation to prove the `DDC` property. For each function, the proof starts with the function precondition as the proof context. We then apply the VST tactics for the current statement of the function body. Each tactic execution updates the proof context by calculating the postcondition of the statement and advances to the next statement, until the end of the function body is reached. At that point, the context should imply the function postcondition.

6 Code Generator Proof

6.1 Property of the code generator

We developed the code generator in Coq, which makes it possible to verify properties of the code generator itself. One interesting property is a generalized `DDC`

property which states that every generated ROS glue code from a valid ROS node model will satisfy the DDC property defined in the Section 5. Intuitively, we should prove that for any input ROS node model, our function template instance satisfies our function specification instance. However, VST tactics can only reason about closed code; it cannot specify properties of our AST templates. Therefore, we cannot directly verify these templates. Instead, we analyze the properties that are required of code generation in order to guarantee the DDC property of the generated code.

The DDC property of generated code states that the destination variable holds the desired value according to the ROS node model before it is used. This DDC property is implied by three code generation properties discussed below. We use the input_glue function from Figure 7 to illustrate how the following three code generation properties imply the DDC property.

```
Definition input_glue_body_statement global_expr control_expr: statement :=
  (Ssequence
    (Sset temp_id global_expr)
    (Sassign control_expr (Etempvar temp_id temp_type)).
```

Fig. 10. Fragment of input glue function body template

Let us first look at the fragment of the template that generates statements in the body of the input_glue function that deliver the value for a single input field. The body is obtained by instantiating the template for each input field. The template has two parameters: global_expr field of message variable and control_expr field of controller input Controller_U. It generates two statements: one copies the message field value (global_expr) to the temporary variable (temp_id); the other sets one field (control_expr) of the controller variable with a temporary variable. We want to show that the DDC property of the input_glue function generated using this template will be satisfied whenever the three properties below hold.

The first code generation property is that the origin (global_expr) and the destination (control_expr) should keep the corresponding relation according to the ROS node model. It ensures that the data is delivered from the right origin to the right destination according to the ROS node model. In this case, global_expr and control_expr in the input_glue function should be consistent with the interface relation. This property guarantees that the Controller_U fields will be assigned by the values from corresponding fields shown in Table 1.

The second property is the valid assignment property, which requires only that the left and right sides of an assignment have the same type. This property implies that the destination variables receive the assigned value after this assignment according to the axiomatic semantics of VST. In this case, Controller_U will hold the value from field x of those three global message variables in Table 1. With the first and second code generation properties, the input_glue function postcondition is guaranteed.

The last code generation property is that the destination variable is not re-assigned by other values before it is used. The third property guarantees that the value of `Controller_U` is preserved until the `Controller_step` function is invoked.

6.2 Proof of the three code generator properties

In this section, we discuss the proof of the three code generator properties presented above. The first property is that we instantiate the `input_glue` function assignment template correctly according to the input ROS node model interface relation. We maintain a list of expressions for each side in the resulting assignments. For the input glue function body, there are lists for `global_expr` and `control_expr`. The first property can be proven by showing that the lists of expressions are consistent with the ROS node model interface relation, as stated by the lemma in Figure 11. In this lemma, `lg_expr` is the list of expressions for `global_expr`, while `lc_expr` is the list of expressions for `control_expr`. The quantified variable `lir` is the list of interface relations from Table 1. To prove the consistency, we verify that the fields of these expression lists are identical to the fields in the interface relation.

```
Lemma relation_consistency_checking :
  forall (lir : list irelation) (lg_expr lc_expr : list expr),
    lg_expr = gen_list_global_variable_expr_input_glue lir →
    lc_expr = gen_list_controller_expr_input_glue lir →
    relation_consistency_checking lir lg_expr lc_expr.
```

Fig. 11. Relation consistency of the input glue function

For the valid assignment property, we only need to check that the lists of types for the left and right sides of the assignment are consistent. The type checking function for the `input_glue` function is shown in Figure 12. Since users may specify an inconsistent ROS node model, mapping a ROS message field with one type to controller input with a different type, the generated assignment can be invalid. The type checking function is applied before generating the `input_glue` function. If type checking returns `FALSE`, ROSGen can set the `error` flag to true and stop generating code. In this way, we guarantee that the generated code always satisfies the valid assignment property.

For the third property, we verify the preservation property by checking that there is no new assignment for the destination variable between the `input_glue` function and `Controller_step` function. This is quite straightforward, because there are no other statements between `input_glue` function and `Controller_step` function in our loop function template. Furthermore, if we were to change our template to add additional statements between the `input_glue` and `Controller_U` calling statements, we would also add the constraint that they do not involve manipulating the `Controller_U` heap.According to the separation logic of VST,

```
Fixpoint type_checking_input_glue
   (ltype_global_fields ltype_controller_fields : list type) : bool :=
  match ltype_global_fields, ltype_controller_fields with
    | [], [] ⇒ true
    |tg:: ltypeg, tc:: ltypec ⇒ andb (type_equal tg tc)
      (type_checking_input_glue ltypeg ltypec)
    |_, _ ⇒ false
  end.
```

Fig. 12. Type checking for input glue function

the value of `Controller_U` is still preserved if those statements manipulate variables in a different heap.

7 Related Work

There has been much work on automatic generation of platform-specific glue code based on the architectural model of the system and the underlying platform specification. In [10,11], code generation for a variety of platforms is performed using AADL models to represent hardware and software architectures and their properties relevant for code generation. None of these papers targeted the ROS platform. More importantly, they do not consider verification of the generated code nor the code generator itself.

There is also a similarity between the intent of our approach and verification of model transformations in domain-specific languages. Most of that work, however, is done in the context of behavioral models, with the goal of ensuring that syntactic constraints are preserved by the transformation [12,13,14]. By contrast, we start with an architectural model, where behavior is implicit, and generate executable code.

8 Conclusions

We propose a verified framework ROSGen for generating glue code for ROS-based control systems. We start with a model of a ROS node capturing external connections of the node and parameters needed to execute the node. The code generator, implemented in Coq, uses this model to instantiate Clight templates and use the VST toolset to reason about the code. We then use CompCert utilities to generate C source code from Clight AST. We discuss how to generalize the proof of data delivery correctness for the generated code to a proof of data delivery correctness for the code generator itself. We apply the approach to the cruise control system for the LandShark robotic vehicle.

Our plans for future work include extending the proof approach to directly reason over quantified Clight templates, allowing for a more natural proof of the code generator correctness. Furthermore, we plan to extend the framework to cover the step function, to be able to reason about control-related properties of the code, in addition to the data delivery properties.

Acknowledgment

Thanks to Andrew W. Appel, Joey Dodds and Qinxiang Cao for help on applying VST and separation logic. We would like to thank the reviewers for their comments that help improve the paper.

References

1. Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.
2. Nicolas Halbwachs. A synchronous language at work: the story of lustre. *Formal Methods for Industrial Critical Systems: A Survey of Applications*, pages 15–31, 2005.
3. Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009.
4. The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
5. Andrew W Appel. Verified software toolchain. In *Programming Languages and Systems*, pages 1–17. Springer, 2011.
6. Wenrui Meng, Junkil Park, Oleg Sokolsky, Stephanie Weirich, and Insup Lee. Verified ros-based deployment of platform-independent control systems. In *University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-MS-CIS-15-01*, February 2015.
7. Xavier Leroy. The compcert c verified compiler, 2012.
8. Andrew W Appel, Dockins Robert, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.
9. Miroslav Pajic, Nicola Bezzo, James Weimer, Oleg Sokolsky, Nathan Michael, George J Pappas, Paulo Tabuada, and Insup Lee. Demo abstract: Synthesis of platform-aware attack-resilient vehicular systems. In *Cyber-Physical Systems (ICCPS), 2013 ACM/IEEE International Conference on*, pages 251–251. IEEE, 2013.
10. Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérome Hugues. Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. In *Reliable Software Technologies: Ada-Europe 2009*, volume 5570 of *Lecture Notes in Computer Science*, pages 237–250. Springer Berlin / Heidelberg, 2009.
11. BaekGyu Kim, Linh TX Phan, Oleg Sokolsky, and Insup Lee. Platform-dependent code generation for embedded real-time software. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, pages 1–10. IEEE, 2013.
12. Anantha Narayanan and Gabor Karsai. Towards verifying model transformations. In *Proceedings of the 5^{th} International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*, pages 191–200, 2008.
13. Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.
14. Levi Lucio and Hans Vangheluwe. Model transformations to verify model transformations. In *Proceedings of the Workshop on Verification of Model Transformations*, June 2013.